WEEK 4

Q- Inorder traversal

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
         vector<int> ans;
        if(root == NULL) return ans;

        stack<TreeNode*> stk;
        TreeNode* node = root;

        while(true){
            if(node !=NULL){
                stk.push(node);
                node = node->left;
            }
            else{
                if(stk.size()==0) break;
                node = stk.top();
                stk.pop();
                ans.push_back(node->val);
                node = node->right;

            }
        }

        return ans;
    }
};
```

Q-Pre order Traversal

```
class Solution {
public:
     void solve(TreeNode* root,vector<int> &ans)
   {
       if(root==NULL)
           return;
       ans.push_back(root->val);
       solve(root->left,ans);
       solve(root->right,ans);
   }

   vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
       solve(root,ans);
      return ans;
   }
};
```

Q-Postorder Traversal

```
class Solution {
public:
     void solve(TreeNode* root,vector<int> &ans)
   {
       if(root==NULL)
           return;

       solve(root->left,ans);
```

```
        solve(root->right,ans);
         ans.push_back(root->val);
    }

    vector<int> postorderTraversal(TreeNode* root) {
         vector<int>ans;
        solve(root,ans);
       return ans;
    }
};
```

Q- Moris Tree Inorder Traversal

```
vector < int > inorderTraversal(node * root) {
  vector < int > inorder;
  node * cur = root;
  while (cur != NULL) {
    if (cur -> left == NULL) {
      inorder.push_back(cur -> data);
      cur = cur -> right;
    } else {
      node * prev = cur -> left;
      while (prev -> right != NULL && prev -> right != cur) {
        prev = prev -> right;
      }

      if (prev -> right == NULL) {
        prev -> right = cur;
        cur = cur -> left;
      } else {
        prev -> right = NULL;
```

```
      inorder.push_back(cur -> data);
      cur = cur -> right;
    }
   }
  }
  return inorder;
}
```

Q- Morris Preorder Traversal

```
vector < int > preorderTraversal(node * root) {
  vector < int > preorder;

  node * cur = root;
  while (cur != NULL) {
   if (cur -> left == NULL) {
    preorder.push_back(cur -> data);
    cur = cur -> right;
   } else {
    node * prev = cur -> left;
    while (prev -> right != NULL && prev -> right != cur) {
     prev = prev -> right;
    }

    if (prev -> right == NULL) {
     prev -> right = cur;
     preorder.push_back(cur -> data);
     cur = cur -> left;
    } else {
     prev -> right = NULL;
     cur = cur -> right;
```

```
    }
   }
  }
  return preorder;
}
```

Q- Left View of Binary Tree

```
class Solution {
public:
   void recursion(TreeNode *root, int level, vector<int> &res)
   {
      if(root==NULL) return ;
      if(res.size()==level) res.push_back(root->val);
      recursion(root->left, level+1, res);
      recursion(root->right, level+1, res);
   }

   vector<int> leftSideView(TreeNode *root) {
      vector<int> res;
      recursion(root, 0, res);
      return res;
   }
};
```

Q- Bottom View Of Binary Tree

```
class Solution {
  public:
   vector <int> bottomView(Node *root) {
      vector<int> ans;
```

```
        if(root == NULL) return ans;
        map<int,int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while(!q.empty()) {
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            mpp[line] = node->data;

            if(node->left != NULL) {
                q.push({node->left, line-1});
            }
            if(node->right != NULL) {
                q.push({node->right, line + 1});
            }

        }

        for(auto it : mpp) {
            ans.push_back(it.second);
        }
        return ans;
    }
}
```

Q- Top View Of Binary Tree

```
class Solution
```

```
{
    public:

    vector<int> topView(Node *root)
    {
        vector<int> ans;
        if(root == NULL) return ans;
        map<int,int> mpp;
        queue<pair<Node*, int>> q;
        q.push({root, 0});
        while(!q.empty()) {
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;
            if(mpp.find(line) == mpp.end()) mpp[line] = node->data;

            if(node->left != NULL) {
                q.push({node->left, line-1});
            }
            if(node->right != NULL) {
                q.push({node->right, line + 1});
            }
            }
                for(auto it : mpp) {
            ans.push_back(it.second);
        }
        return ans;
    }
```

```
};
```

Q- Preorder Postorder Inorder in a single Traversal

```
#include <bits/stdc++.h>
using namespace std;
struct node {
  int data;
  struct node * left, * right;
};
void allTraversal(node * root, vector < int > & pre, vector < int > & in , vector < int > & post) {
  stack < pair < node * , int >> st;
  st.push({
    root,
    1
  });
  if (root == NULL) return;
  while (!st.empty()) {
    auto it = st.top();
    st.pop();
    if (it.second == 1) {
      pre.push_back(it.first -> data);
      it.second++;
      st.push(it);

      if (it.first -> left != NULL) {
        st.push({
          it.first -> left,
          1
        });
```

```
    }
   }
   else if (it.second == 2) {
    in .push_back(it.first -> data);
    it.second++;
    st.push(it);

    if (it.first -> right != NULL) {
     st.push({
      it.first -> right,
      1
     });
    }
   }
   else {
    post.push_back(it.first -> data);
   }
  }
}
```

Q- Vertical Order Traversal

```
#include <bits/stdc++.h>

using namespace std;

struct node {
  int data;
  struct node * left, * right;
};
```

```
vector < vector < int >> findVertical(node * root) {
 map < int, map < int, multiset < int >>> nodes;
 queue < pair < node * , pair < int, int >>> todo;
 todo.push({
  root,
  {
   0,
   0
  }
 });
 while (!todo.empty()) {
  auto p = todo.front();
  todo.pop();
  node * temp = p.first;
  int x = p.second.first, y = p.second.second;
  nodes[x][y].insert(temp -> data);

  if (temp -> left) {
   todo.push({
    temp -> left,
    {
     x - 1,
     y + 1
    }
   });
  }
  if (temp -> right) {
   todo.push({
```

```
      temp -> right,
      {
       x + 1,
       y + 1
      }
     });
   }
  }
  vector < vector < int >> ans;
  for (auto p: nodes) {
   vector < int > col;
   for (auto q: p.second) {
    col.insert(col.end(), q.second.begin(), q.second.end());
   }
   ans.push_back(col);
  }
  return ans;
}
```

Q- Root to Node Path in a Binary Tree

```
bool getPath(node * root, vector < int > & arr, int x) {
  if (!root)
   return false;
  arr.push_back(root -> data);
  if (root -> data == x)
   return true;
  if (getPath(root -> left, arr, x) ||
   getPath(root -> right, arr, x))
   return true;
```

```
  arr.pop_back();
  return false;
}
```

Q- Check if Binary Tree is the mirror of itself or not

```
void mirror(Node* root)
{
   if (root == NULL)
      return;

   queue<Node*> q;
   q.push(root);

   while (!q.empty())
   {

      Node* curr = q.front();
      q.pop();

      swap(curr->left, curr->right);

      if (curr->left)
         q.push(curr->left);
      if (curr->right)
         q.push(curr->right);
   }
}
```

Q- Check for children sum Property

```
oid reorder(node * root) {
  if (root == NULL) return;
  int child = 0;
  if (root -> left) {
    child += root -> left -> data;
  }
  if (root -> right) {
    child += root -> right -> data;
  }

  if (child < root -> data) {
    if (root -> left) root -> left -> data = root -> data;
    else if (root -> right) root -> right -> data = root -> data;
  }

  reorder(root -> left);
  reorder(root -> right);

  int tot = 0;
  if (root -> left) tot += root -> left -> data;
  if (root -> right) tot += root -> right -> data;
  if (root -> left || root -> right) root -> data = tot;
}
void changeTree(node * root) {
  reorder(root);
}
```